Programming in the Era of AI

Daniel Lemire, professor
Université du Québec (TÉLUQ)
Montréal 🇨🇦

blog: https://lemire.me

X: @lemire · GitHub: github.com/lemire

TODO

- add https://lemire.me/benchmarks/ada_history/index.html https://claude.ai/code/artifact/ce1fe8b4-f5eb-4732-9cdd-a8437a36d99b
- information limit https://x.com/lemire/status/2091229055802634669?s=46&t=-zo9kVFDyKuN4X1cdtkIrw
- generic software -> specialized software (my own web framework, my own video recording software)
- verfiable vs non-verifiable, Cette notion de ce qui est vérifiable et de ce qui est difficilement vérifiable.
- more performant software (bar higher), more correct software

Where I am coming from

- Author of high-performance libraries integrated into major browsers, runtimes and standard libraries: **simdutf**, **fast_float**, **simdjson**, **Roaring Bitmaps**.
- Among the top 2% of scientists worldwide (Stanford/Elsevier), 100+ peer-reviewed papers.
- Among the 1000 most-followed developers on GitHub.
- **And: I have not typed most of my code by hand in over a year.**

Preface

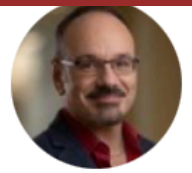
Back in the 1990s

The singularity

> *« I believe that the creation of greater than human intelligence will occur during the next thirty years. (I'll be surprised if this event occurs before 2005 or after 2030.) »*
>
> Vernor Vinge, 1993

Part 1

Where we actually are



Daniel Lemire
@lemire

Boost

What fraction of your code is typed by hand (no AI)?

none (vibe coding)	21.7%
20%	22.5%
50%	19.2%
all (old school)	**36.7%**

1,023 votes · Final results

7:19 PM · Mar 15, 2026 · **13.5K** Views

Daniel Lemire
@lemire

Boost

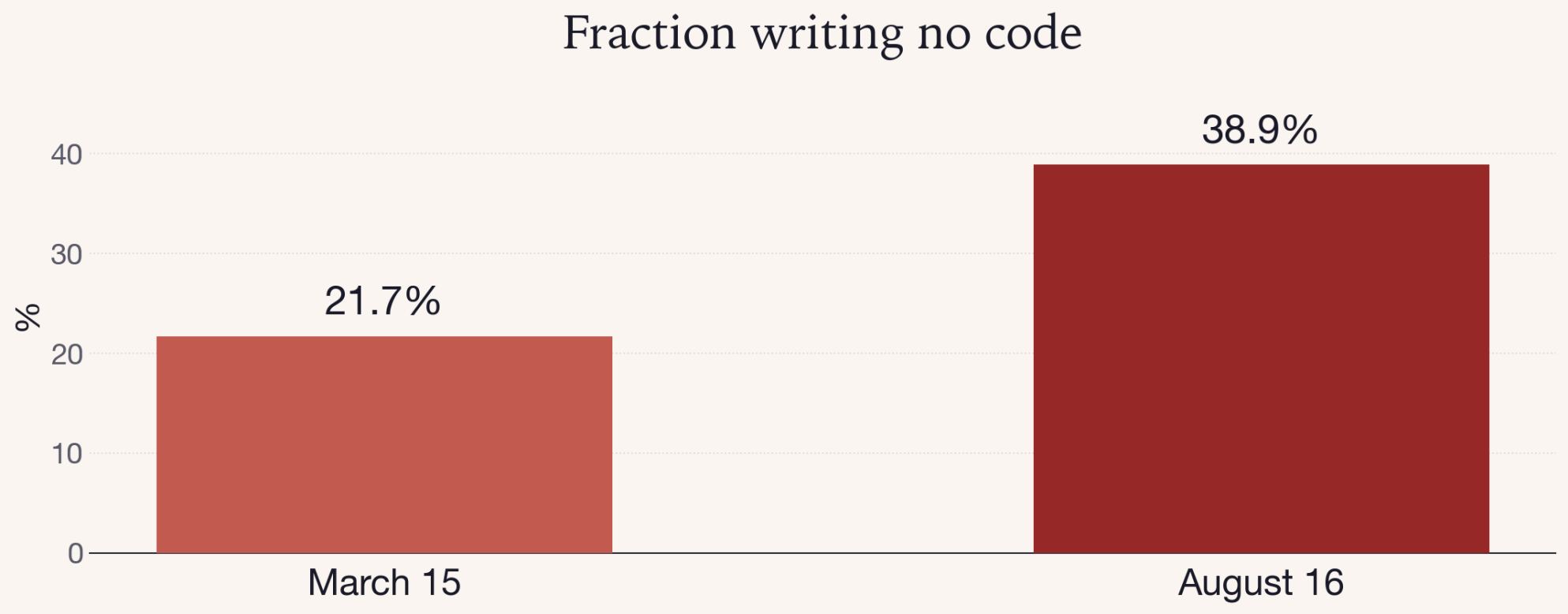
What fraction of your code is typed by hand (no AI)?

none (it is the future)	**38.9%**
20%	21.6%
50%	14.8%
all (old school)	24.6%

943 votes · Final results

2:58 PM · Aug 16, 2026 · **6,001** Views

What that poll really shows

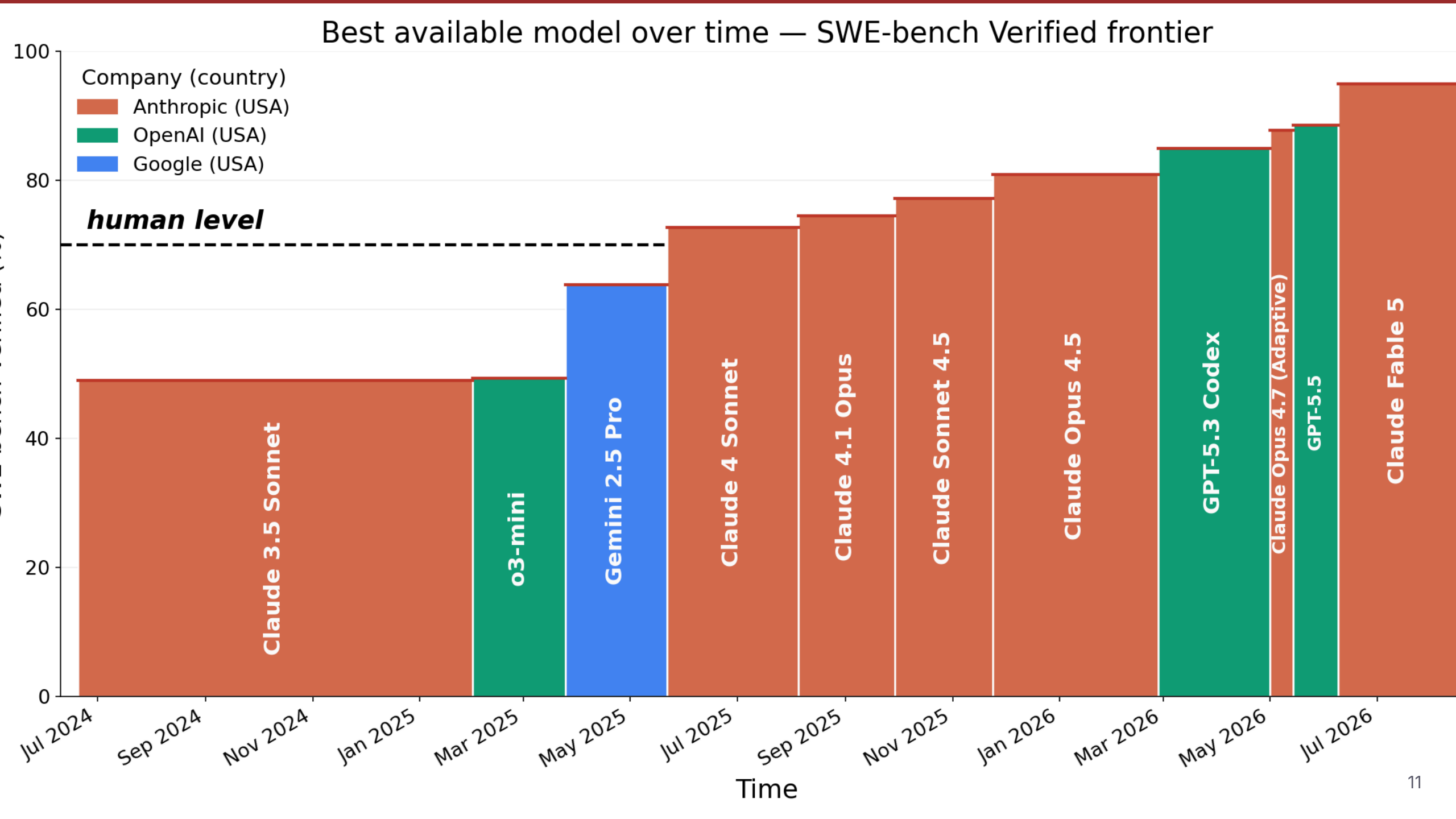


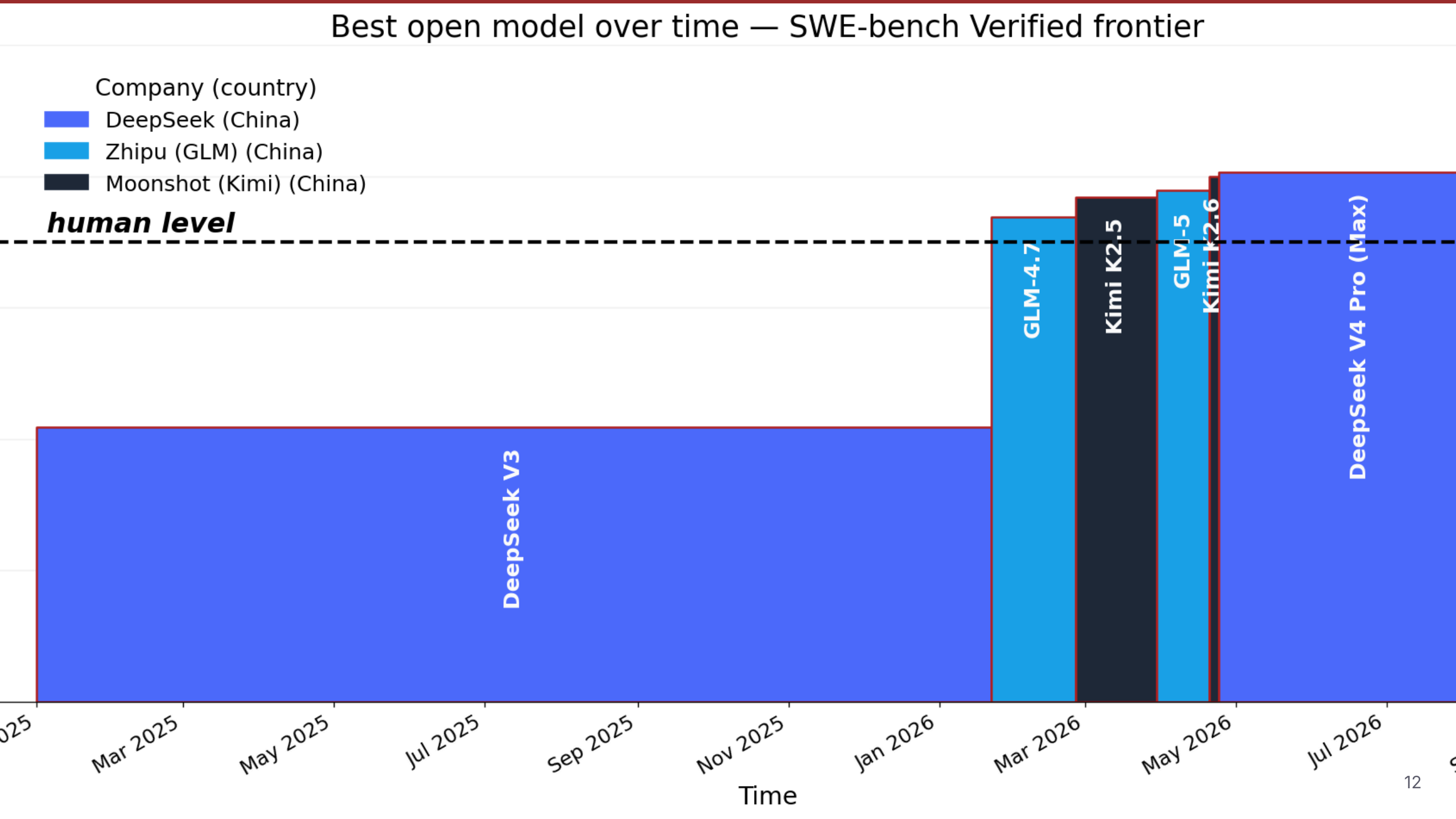
- this is a practice change, not a tool upgrade.

SWE-bench Verified

- Real GitHub issues from real open-source Python repositories.
- The model gets the repository, the issue text, and nothing else.
- Success = the project's own **hidden tests** pass afterwards.
- 500 tasks, human-validated as solvable.

It is not a quiz. It is a work order.


Best available model over time — SWE-bench Verified frontier
Company (country)
Anthropic (USA)
OpenAI (USA)
Google (USA)
human level
100
80
60
40
20
0
Claude 3.5 Sonnet
o3-mini
Gemini 2.5 Pro
Claude 4 Sonnet
Claude 4.1 Opus
Claude Sonnet 4.5
Claude Opus 4.5
GPT-5.3 Codex
Claude Opus 4.7 (Adaptive)
GPT-5.5
Claude Fable 5
Jul 2024
Sep 2024
Nov 2024
Jan 2025
Mar 2025
May 2025
Jul 2025
Sep 2025
Nov 2025
Jan 2026
Mar 2026
May 2026
Jul 2026
Time

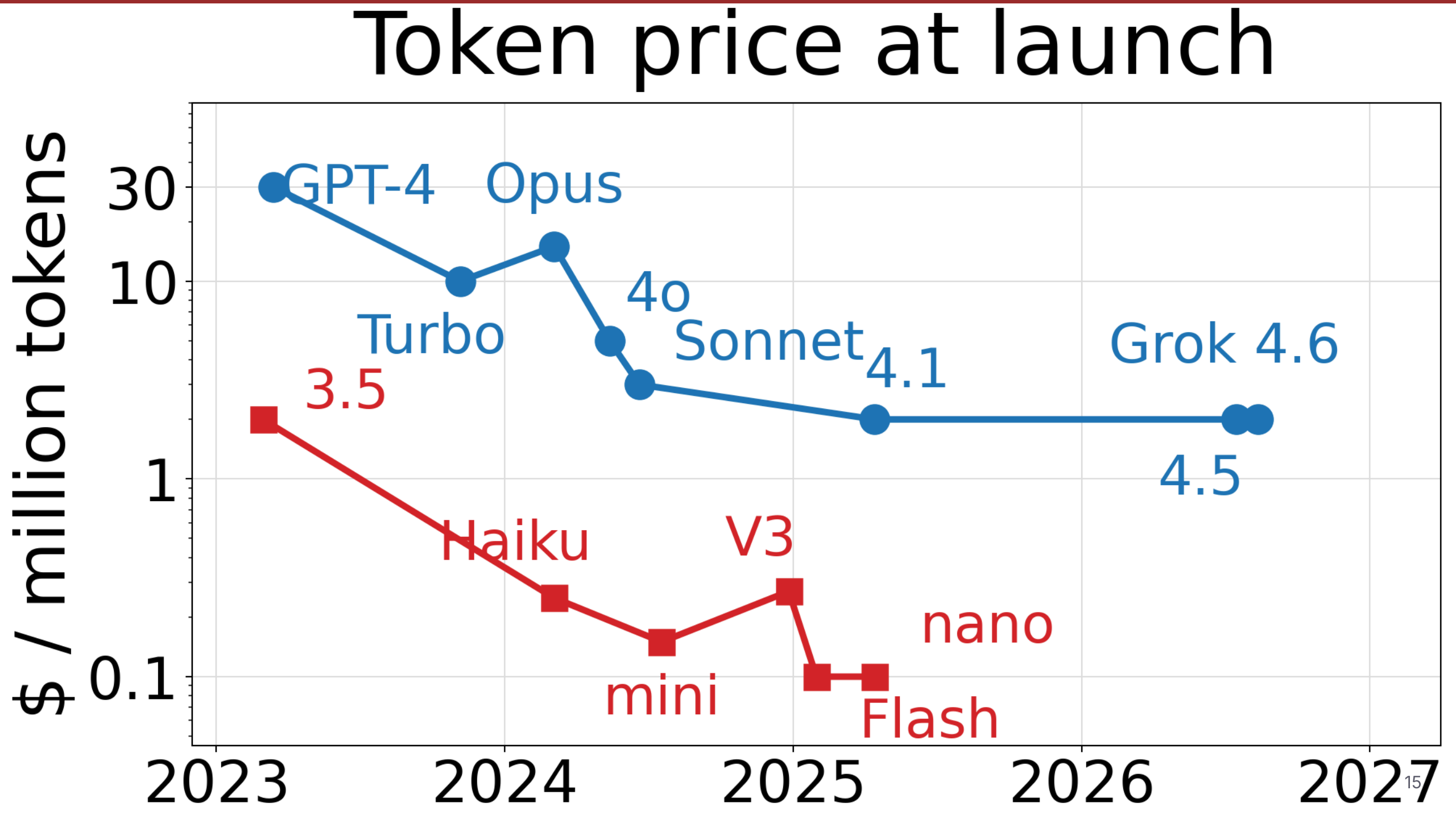

Best open model over time — SWE-bench Verified frontier
Company (country)
DeepSeek (China)
Zhipu (GLM) (China)
Moonshot (Kimi) (China)
human level
DeepSeek V3
GLM-4.7
Kimi K2.5
GLM-5
Kimi K2.6
DeepSeek V4 Pro (Max)
Mar 2025
May 2025
Jul 2025
Sep 2025
Nov 2025
Jan 2026
Mar 2026
May 2026
Jul 2026
Time

Superhuman

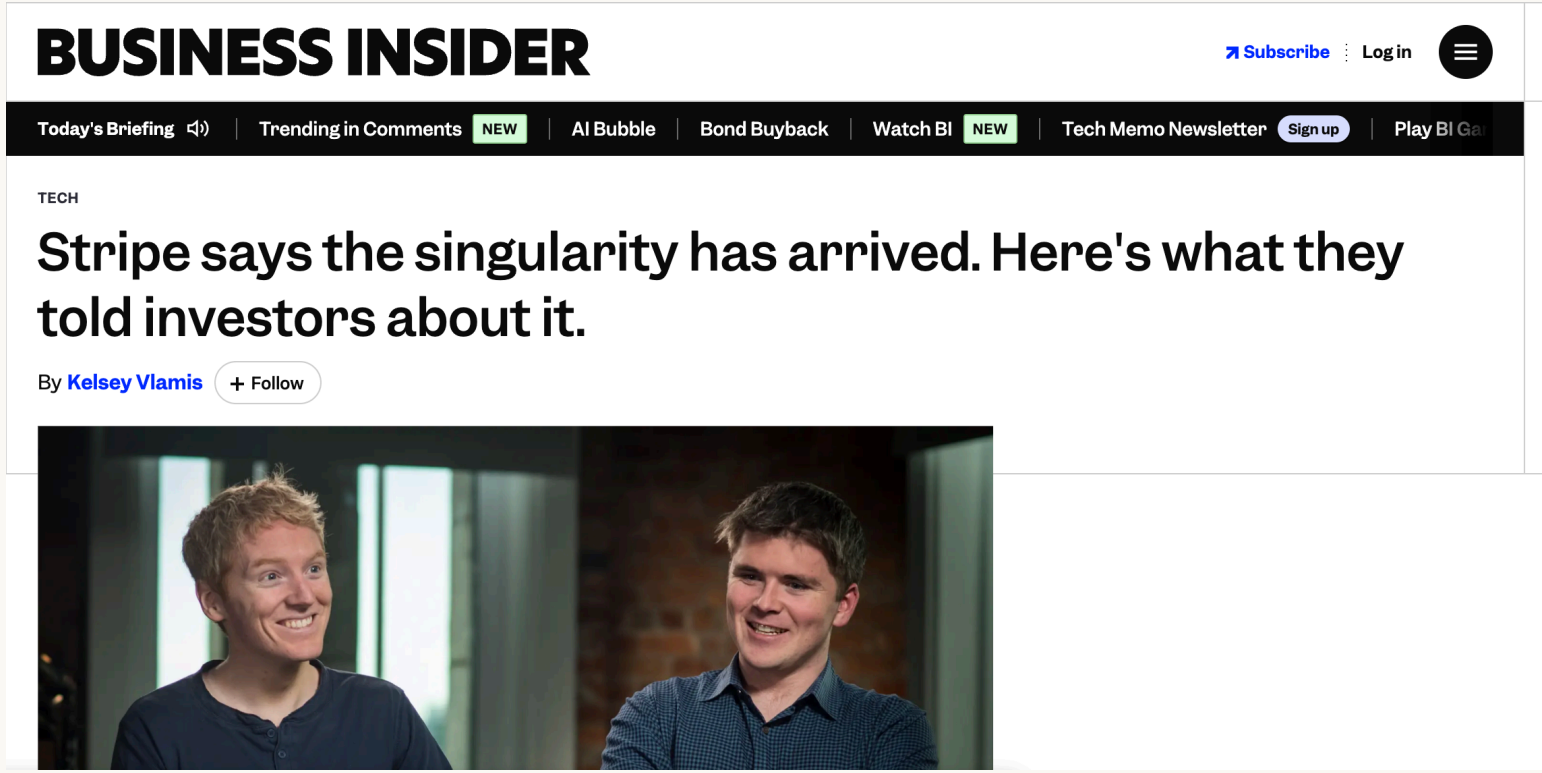
- Top systems now resolve **~90%** of SWE-bench Verified tasks.
- A skilled human given the same isolated issue, the same repository, and no colleagues, does **not** do better.
- The machine does it in minutes, in parallel, for a few dollars.

Open weights are close behind

- The leading open-weight models sit within a few points of the leading proprietary models.
- They run on hardware you can own, or rent by the hour.


Token price at launch
$ / million tokens
30
10
1
0.1
2023
2024
2025
2026
2027
GPT-4
Opus
Turbo
4o
Sonnet
4.1
Grok 4.6
4.5
3.5
Haiku
V3
nano
mini
Flash

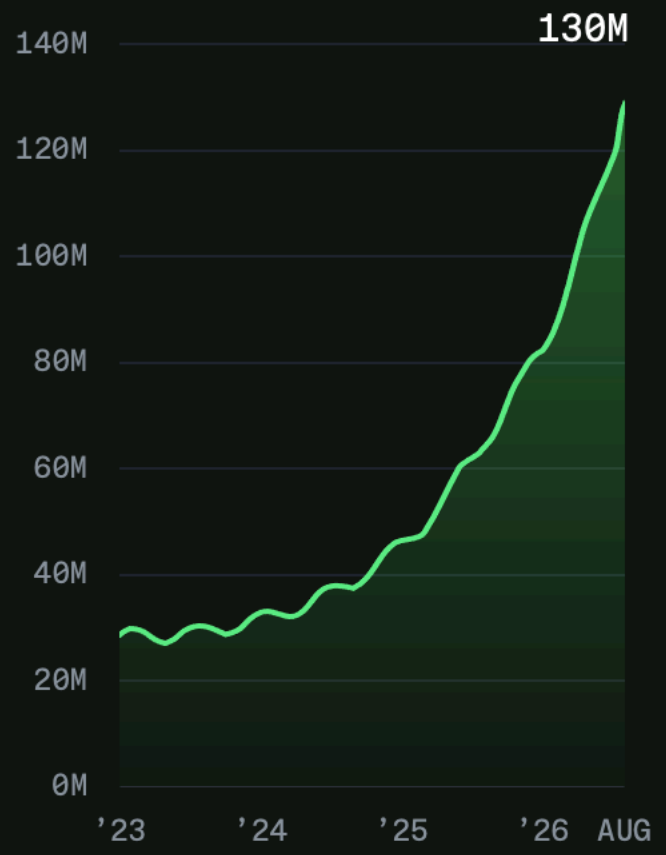
> *January 1st marked the beginning of the singularity.*
>
> —Stripe, August 19, 2026



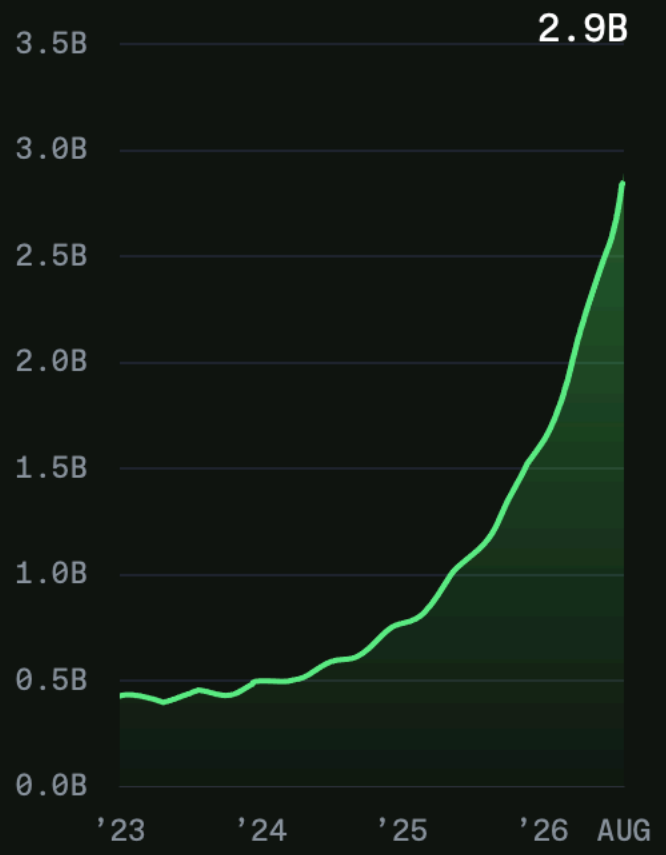
People often talk about this concept called AGI, meaning artificial general intelligence, that is, an AI as intelligent as a person. I actually think we crossed that threshold about three months ago.

— Marc Andreessen, May 19, 2026

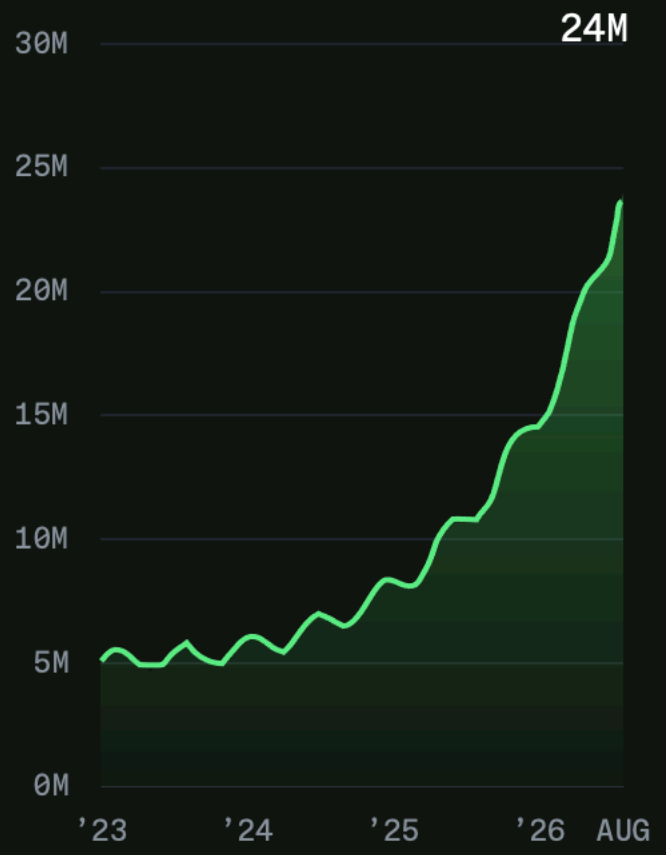
Merged pull requests per month



Commits per month



New repos per month

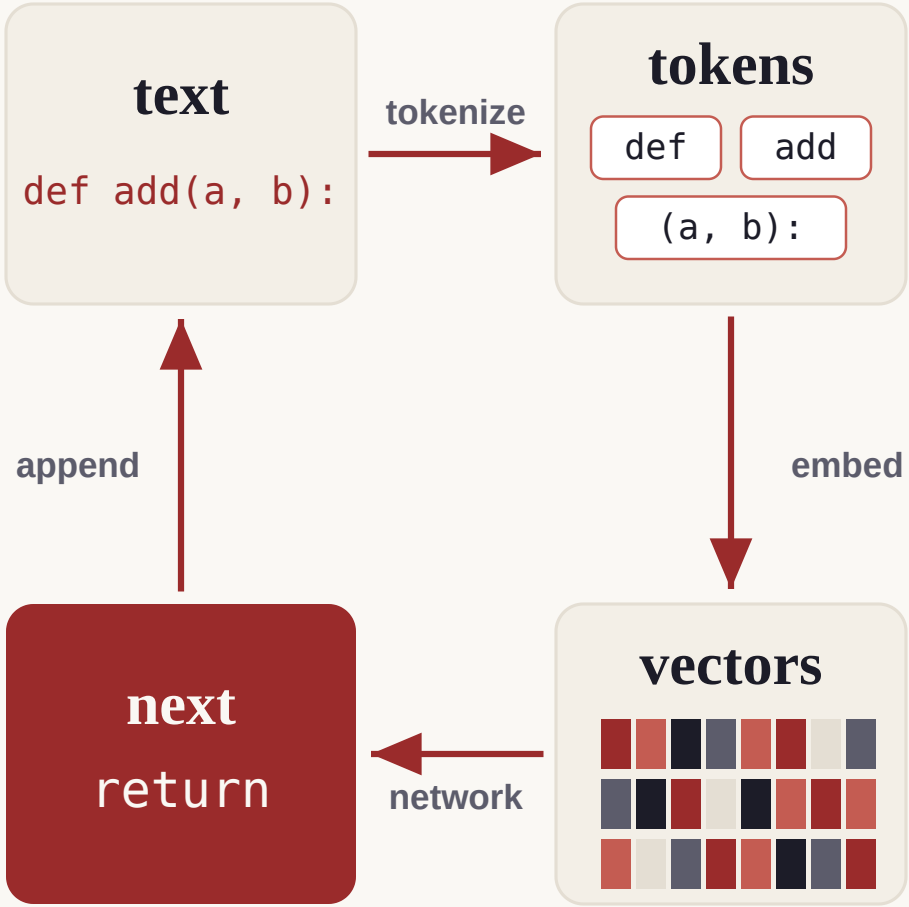


Classified as Microsoft Confidential

Part 2

What makes an agent work

How an LLM works



Tokens in, tokens out

- Text is chopped into **tokens**: words, pieces of words, punctuation.
- Each token becomes a **vector** in a high-dimensional space.
- The network looks at those vectors and picks a likely next token.
- Sample one. Append it. Repeat.

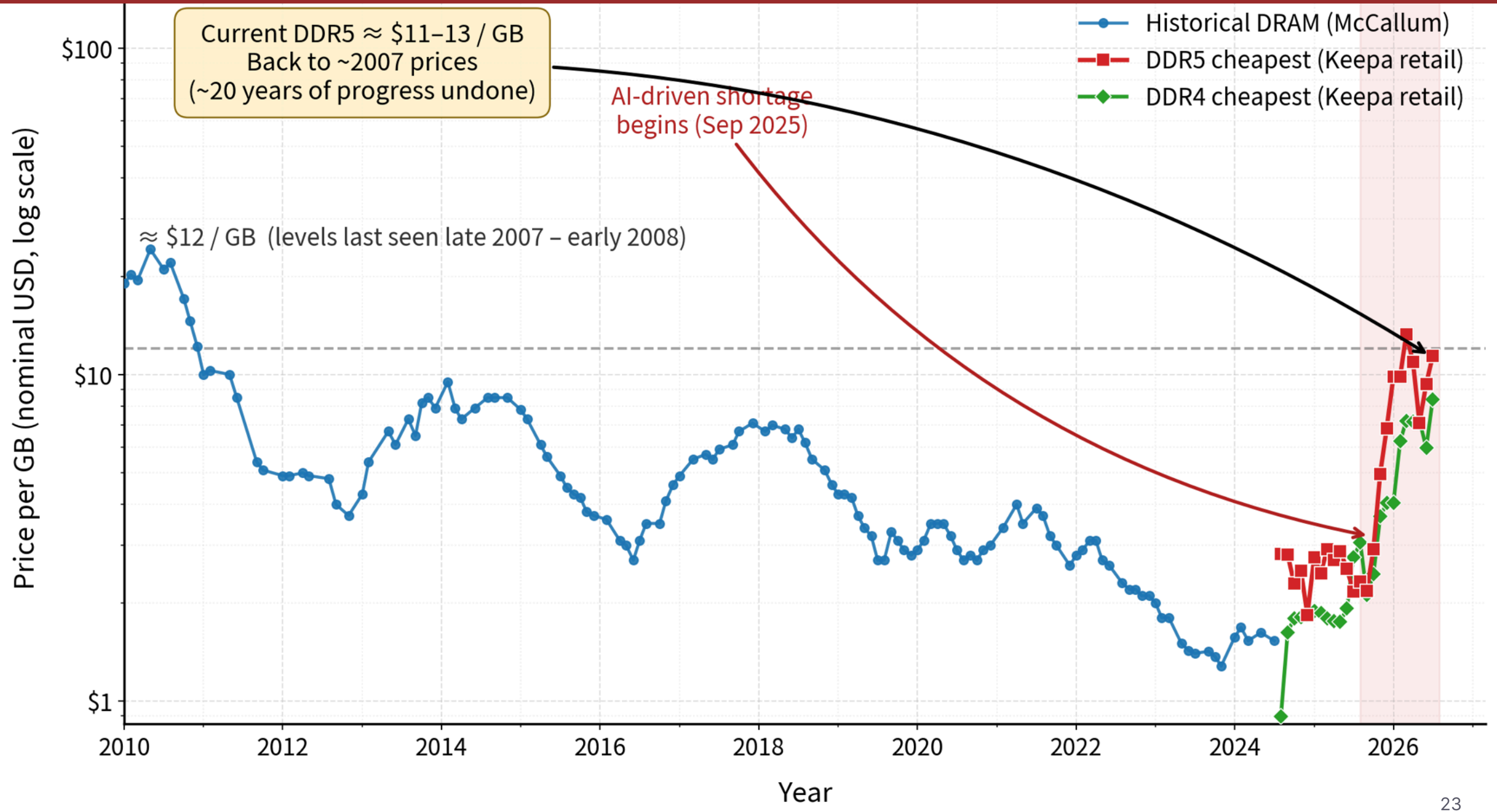
That is the entire machine.

The weights

0.5 GB per billion parameters at 4-bit. Active parameters are computed per token; the **total still sits in RAM**.

MODEL	LAB	TOTAL / ACTIVE	4-BIT	LICENSE
Qwen3.8 Max	Alibaba	2.4T / ~95B	1.2 TB	Custom
Kimi K3	Moonshot	2.8T / ~104B	1.4 TB	Custom
GLM-5.2	Zhipu	753B / ~40B	380 GB	MIT
DeepSeek V4 Pro	DeepSeek	1.6T / ~49B	800 GB	MIT
MiniMax M3	MiniMax	428B / ~23B	210 GB	Community

Closed models (Claude, GPT, Grok) do not publish the number.



Sources: John C. McCallum historical dataset (via Stanford DAM) + Keepa Amazon retail prices for cheapest consumer modules. Nominal USD. Data through July 2026.

How can a language model write software?

- I write a query, it is maybe 80 characters.
- I get back a full program with 100,000 lines.
- Not how it works, there is more information invovled.

The inference loop

```
while not done:
    action = model(context)     # tokens out
    result = tool(action)       # shell, tests, git
    context += result           # tokens in
```

Every serious coding product ships this loop.

That is why agents can code

- **Write** a patch as tokens.
- **Run** the tests as a tool.
- **Read** the failure as tokens.
- **Try again.**

Agentic

The model is no longer answering. It is **acting**.

- call a tool
- read the result
- decide the next step
- repeat

Chat completes a sentence. An agent completes a **job**.

The loop is dumb

Same loop, different models → wildly different SWE-bench scores.

A weak model in the loop thrashes:

- misreads failures
- patches symptoms
- oscillates between two wrong fixes

Reinforcement learning

act, get graded, update.

```
trajectory = model acts in environment   # patch, proof, tool calls
reward     = verifier(trajectory)        # tests pass? answer correct?
weights   += learn from reward           # reinforce what worked
```

- RL rewards **outcomes** — trajectories no human ever wrote.

The trick: put the loop in the training

Code has a rare property: the reward is **mechanically verifiable**.

Tests pass or they don't. The build compiles or it doesn't.

Reinforcement learning on verifiable rewards (RLVR):

- run the model in the loop, on real repositories
- reward trajectories that converge to green tests
- update the weights

The model no longer learns to predict plausible code.
It learns to **make the loop converge**.

Constraints

Benchmarks and tests as the steering wheel

The central problem

A language model will produce something plausible **every single time**.

It has no way, on its own, to know whether it is right.

So the entire engineering problem becomes: **give it something it can check.**

The hierarchy of constraints

From weakest to strongest:

1. "Looks good to me"
2. A type checker or linter
3. A unit test
4. A **property** test or differential test against a reference
5. A **fuzzer**
6. A **benchmark** with a numeric target

Each level lets you delegate more and supervise less.

Tests are no longer just for regressions

- Historically: tests protect code you already wrote.
- Now: tests **specify** code that does not exist yet.
- The test suite is the contract that the agent iterates against.

Write the test first — not for purity, but because it is the only instruction the machine cannot talk its way out of.

A benchmark is an even better constraint

A test says *correct / incorrect*.

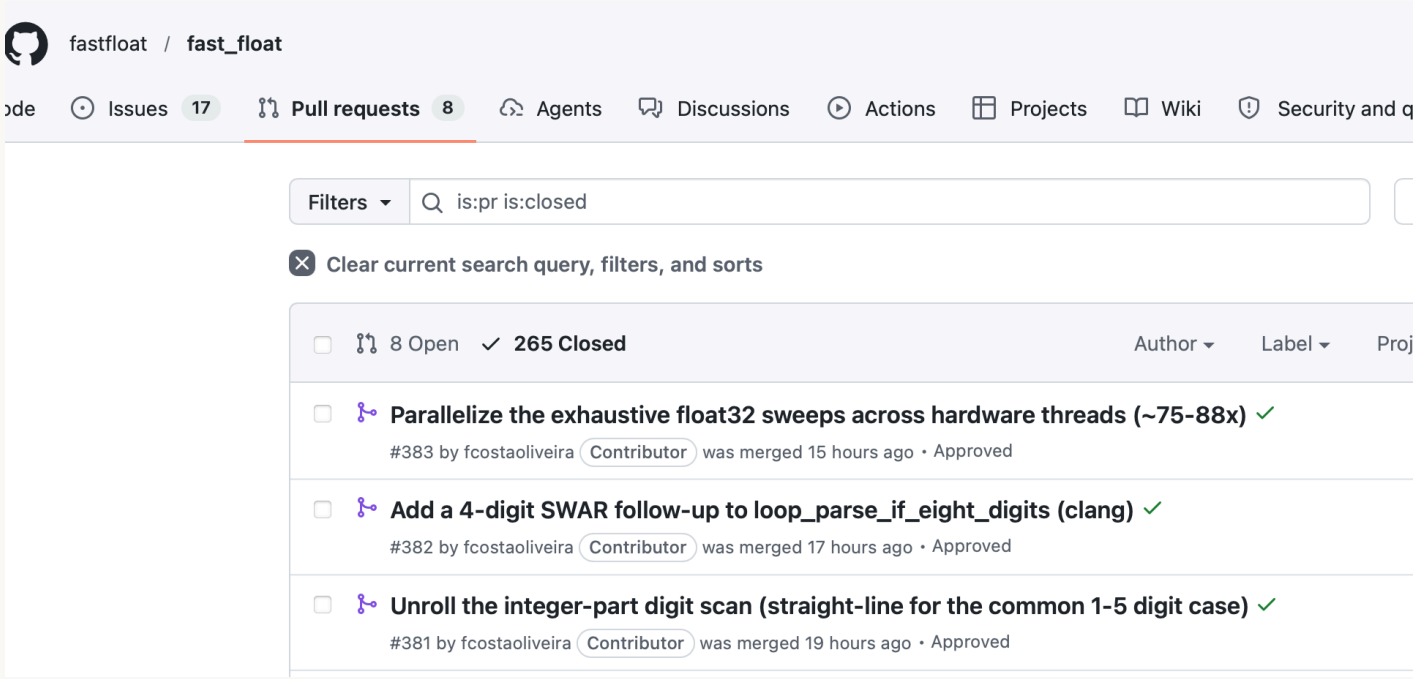
A benchmark says *how much better*.

```
Make `parse_number` faster. Run `make bench` after every
attempt. Do not stop until cycles/byte drops below 0.20
and the differential fuzzer still passes.
```

The agent now has a gradient to climb, and a stopping condition.

It works on hard code

- `fast_float` is used by Chrome, Safari, GCC, Rust, Go, MySQL.
- It has been tuned by hand for five years.
- An agent, given the benchmark, found **two independent 10% improvements**.



Why that surprised me

- I did not believe there was 20% left.
- The agent was not smarter than the contributors. It was **more patient**.
- It tried hundreds of variants overnight and kept the ones the benchmark liked.

Search plus a good objective function beats intuition.

Differential testing: the workhorse

```
Here is a slow, obviously-correct reference implementation.
Here is a fuzzer that feeds random inputs to both and
compares outputs, including edge cases: empty input,
one byte, unaligned buffers, invalid encodings.

Write a fast version. Run the fuzzer after every change.
```

> You are not reviewing the code. You are reviewing **the contract**.

> *If I cannot state how I would check the result, I do not delegate it.*

Context

The scarce resource

Context windows are large now

A million tokens is roughly a 3000-page book. So the problem is solved?

No. A large context is a large *haystack*.

- Attention dilutes: the more irrelevant material you include, the more the model misses the relevant part.
- Stale information competes with fresh information — and looks identical.
- Cost and latency scale with what you keep.
- The failure is silent: you get a confident answer built on the wrong file.

Context management: the moves

1. **Curate, don't dump.** Retrieve the three right files, not the repository.
2. **Persist the durable facts** in a file the agent always reads.
3. **Externalize state** to disk and to Git, not to conversation history.
4. **Isolate**: give a sub-task its own fresh context.
5. **Compact deliberately**: summarize and restart rather than letting a session rot.

AGENT.md

Put the rules you are tired of repeating in a file the agent **always** reads.

```
# AGENT.md

- After every C++ change, run `clang-format`.
- Tests must pass under ASan and UBSan.
- Do not invent APIs. If it is not in the tree, it does not exist.
```

Grok: `AGENT.md` / `AGENTS.md`. Claude: `CLAUDE.md`.

Commit it. Then every session, every teammate, every subagent starts from the same contract.

Orchestration

Many agents at once

git worktree: two directories, one repository

```
git worktree add ../feat-a -b feat-a
```

Each checkout has its own files and its own branch.
The objects live in one `.git`.

Two agents editing `src/foo.c` in the **same** working tree overwrite each other.
A worktree is the isolation primitive. Parallelism is why you need it.

Why parallelism, concretely

- A single agent run is minutes of wall-clock time, mostly waiting on a build or a test suite.
- Four agents cost four times the tokens and roughly **zero extra minutes**.

Worktrees keep them from colliding on disk.
The next question is *what kind of parallel*: a **child**, or a **new conversation**.

Subagent vs new session

A **subagent** is a child **inside this conversation**.
It has its own context window. It reports a **summary** back.

A **new session** is a **separate conversation**.
Nothing comes back unless you copy it.

Both can sit in a worktree. They are not the same thing.

Commands

	NEW SESSION	SUBAGENT
Grok	`/new` · `grok --worktree=feat "…"`	"spawn a subagent to review this"
Claude	`/clear` · `claude --worktree feat`	"use a subagent to ..." · `@explore`

A worktree isolates **files**. A subagent isolates **context**. You can combine them.

Context window

Subagent

- fresh window for the dirty work: grep, logs, failing tests
- the parent keeps the plan
- only a **summary** is written back into the parent

New session

- also a fresh window
- **nothing** is written back automatically
- you are the merge, in Git or by paste

When the subagent is better

Use a subagent when the parent still needs the result **in this conversation**.

- find every call site — do not dump forty files into my context
- run the tests — tell me what failed
- review this patch — return findings

The parent stays the coordinator.
The child is disposable context.

When a new session is better

Start a new session — usually in a worktree — when the work is a **job**, not a lookup.

- a second feature, a second PR
- hours of iteration you do not want in this transcript
- a model that should not inherit this session's wrong assumptions

You will merge in Git, not in the chat.

Trade-offs

	SUBAGENT	NEW SESSION
Reports back	yes, a summary	only if you copy
Shares your plan	yes	no
Pollutes parent context	little	none
File isolation	optional worktree	you pick the directory
Lifetime	dies with the task	you resume it

Default: subagent for a question. New session for a branch of work.

Pattern 1: fan-out over independent work

One task list, one agent per item.

- Migrate 30 files to a new API
- Add tests to 12 modules
- Port a kernel to five instruction sets

Requirement: the items must not touch the same files.

Pattern 2: the panel

Ask **N** agents the same question independently, then compare.

- Three designs for the same feature, then pick one.
- Three reviewers on the same patch, keep findings that two of them agree on.

Independent samples catch what one confident sample does not.

Pattern 3: adversarial verification

The generator and the critic must not be the same context.

```
Agent A: implement the feature until the tests pass.
Agent B: given only the diff, try to prove it is wrong.
         Find an input where it misbehaves.
```

A model asked to defend its own work will defend it.

What does *not* parallelize

- Anything with a shared, mutable, non-mergeable resource: one database, one port, one flaky integration test.
- Work where step 2 genuinely needs the answer to step 1.
- Anything where merging costs more than doing.

And you still have to read the results. That does not parallelize at all.

Part 3

What we must change

Teaching: what stops working

- Take-home programming assignments as an assessment of *coding*.
- "Implement a linked list" as a filter.
- Grading the artifact instead of the reasoning.
- Any exercise whose specification is complete enough to paste into a chat box.

If a task is fully specified, it is already automated.

Teaching: what becomes essential

- **Reading** code critically — the skill formerly gained by writing it.
- **Specification**: turning a vague need into a checkable statement.
- **Testing**, fuzzing, property-based thinking.
- **Measurement**: benchmarks, statistics, performance counters.
- **Systems fluency**: the shell, Git, build systems, deployment.
- Knowing what is **hard**, so you notice when the answer is too easy.

Assessment has to move

- Oral defence of a design decision.
- Review a patch, in the room, and justify the review.
- Give students an agent and a hard problem, and grade the **process**.
- Assume the tool is present, then ask for something it cannot do alone.

We stopped grading arithmetic when calculators arrived. We did not stop teaching mathematics.

Work organization is the real bottleneck

Two roles are emerging:

- **The coder-analyst** — closest to the problem, now able to build the tool themselves.
- **The architect-programmer** — sets constraints, defines interfaces, owns the tests, reviews.

The team structure built around "one specification, thrown over a wall, then six months of implementation" no longer matches the cost structure.

Review becomes the constraint

- Generating a 2000-line patch costs minutes.
- Reviewing a 2000-line patch costs a day.
- The queue moves to the humans.

Everything that makes review cheaper is now the highest-value engineering investment: small diffs, strong tests, clear invariants, automated checks.

Research: what to prioritize

- **Verification at scale**: how do we gain confidence in code no human read line by line?
- **Better objective functions**: benchmarks that capture what we actually want.
- **Context and retrieval**: what to show a model, and when.
- **Orchestration**: when does a second agent help, and when is it noise?
- **Empirical software engineering**: we have a new, measurable subject and very little data.

The bad news

- Models are confident when they are wrong, and the failure is invisible in the diff.
- Generated code raises volume: more code, more surface, more supply-chain exposure.
- Prompt injection is a real attack: an agent that reads the web and can run commands is a new class of target.
- Skills are eroding in the people who never had to acquire them the hard way.
- The economics of junior positions are genuinely under pressure.

None of this is a reason to opt out. All of it is a reason to build the checkers.

If you take five things home

1. The capability is real, it is cheap, and it is available to your competitors.
2. **Whatever you cannot check, you cannot delegate.** Build the checkers first.
3. Tests and benchmarks are no longer overhead — they are the steering wheel.
4. Context is the scarce resource: curate, persist, isolate, compact.
5. Parallel agents buy throughput only when verification is automated.

Questions?

Daniel Lemire — lemire.me (blog)

https://simdjson.org · https://roaringbitmap.org

https://simdutf.github.io/simdutf/ · https://fastfloat.github.io/fast_float/